Final Report for Project Joseph Hoane Season 2

By: Aiden Streicher, Andrew Drost, Benjamin Johll, Janus Golczewski

Client: Bowen Weng

Course: COMS 4020 Senior Design

Instructor: Simantra Mitra

TA: Mohammed Rahman

Fall 2025

Chapter 1: Requirements

1.1 Purpose

Our task was simple. Given a chess board, a camera and robot arm, our goal was to make the robot play chess. This is broken down into a variety of sub-tasks which include:

- Chess Simulation: Implementing a chess engine (stockfish) into a game of chess so it can make moves. We also need to update the chess board with a FEN string input.
- Piece Recognition: Our camera needs to look at the pieces and recognize both what piece it is (color, type) and what square of the board it is on. From this it needs to generate a FEN string of the board state used to update the board.
- Robot Movement: The robot arm needs to be able to hover and pick up the piece being moved, and drop it at its new location. It also needs to be able to handle more complex moves such as a capture.

These 3 core goals are what leads to the intended end result. Chess is an old game which has been around for centuries. Robotics is a relatively new discovery. Combining the two is an exciting prospect that pushes the limits of technology.

1.2 Definitions

UR10e- The Universal Robotics arm we used to move pieces.
2f85- The gripper the robot uses to pick up the pieces.
ROS- The Robot Operating System, used to operate the robot.
MuJoCo- A simulator used to run simulations on a UR10e_2f85 robot.
RabbitMQ- Queue to decouple image processing pipeline from the rest of the architecture.
Ubuntu 20.04- OS of laptop controlling the robot. Required for using ROS Noetic.
Docker- A containerization platform for isolating components and provisioning services.
Stockfish- The chess engine used to determine the robots moves.

PostgreSQL- Database used to persist some data like moves and game metadata
Minio- Eases the sharing of media between services.
Golang- A programming used to implement the core worker/orchestrator service for this project.
Python/FastAPI- Python framework to create HTTP API endpoints.
Node.js- Javascript runtime used to run the chess corner detection model with Tensorflow JS
Vue.js- A frontend framework for creating a GUI to interact with the rest of the software
Ultralyitics/Yolo- Used to train image recognition models and to recognize pieces.
OpenCV- A library for tasks related to computer vision.
gRPC- A modern RPC framework for remote calls between different services
HTTP- A protocol for communication between different services.

1.3 Constraints

Laptop Environment

The laptop our code must run on has no GPU and cannot import many libraries limiting the resources we could use on our project.

An example of this is that rather than using an analytical inverse kinematics library, such as IKFast, we created our own in-house solution.

Gripper Malfunctions

During the project there was an issue with the gripper frequently not responding to commands from code. As such we controlled it manually during our testing and demo.

Lab Hours

The lab was open 3 days a week from 4-6 PM. This gave us limited time to actually test with the real robot.

Chapter 2: Design

2.1 Board Detection

Purpose:

In order to determine the proper moves, we must understand the state of the board. Using machine learning and image processing techniques on a camera feed, we can accurately detect the board and its configuration.

2.1.1 Piece Detection

We use the YOLOv8 detection model to identify chess pieces on the board. The model was trained on many images of chessboards and pieces that were labeled in label studio. The ML model outputs a list of discovered pieces, their location on the image, and the type of piece it is with a confidence percentage.

2.1.2 Perspective Warp

To translate the pixel coordinates of each piece in the image into an actual position on the board, we perform a top-down perspective warp which results in a perfectly square image containing only the chess board. A homogenous transformation matrix is used to translate between the original and the warped images. After the warp, the board can easily be divided into an 8x8 grid that piece locations can be mapped to.

2.1.3 Corner Detection

Creating the perspective warp matrix requires knowing the location of the four chessboard corners in the original image. We use a machine learning model created by Github user Elucidation in order to find the general location of the corners. Afterwards we can perform an initial warp that gives a top down view of the chess board.

2.1.4 Line Detection

The corners found through the machine learning model are almost never precise enough to accurately translate piece locations. In order to find the most accurate corners possible, we use line detection to find the edges of the board and use the intersection of the edges as the corners for the final warp matrix. This process utilizes the canny edge detector to highlight the lines in the image, and then uses the hough lines algorithm to find the largest lines in the image, corresponding to the chessboard edges and grid lines.

This process can also detect lines that are not useful to us, so we must do some filtering in order to find only the lines that are a part of the board. One method for this is sorting the lines by angle, into horizontal and vertical groups. Any lines that have an invalid angle are discarded. Another method we utilize is checking the distance between each line. Because of the even grid spacing in a chessboard, each pair of lines should have the same amount of distance between them. By calculating the median distance between each line, we can determine the proper spacing and remove the lines that don't fit within a threshold.

2.1.5 Image Difference

Occasionally the machine learning model for detecting chess pieces will misclassify a piece or not recognize one. To handle these cases, we use image subtraction to compare the previous image with the newly obtained image. This allows us to find the location of the chess piece that just moved. Because only one piece can move at a time, once we know what the board state is, we can keep track of it utilizing the image difference method.

2.2 Orchestrator

Purpose:

Our software has a number of components, we need something to tie everything together and to be the "brain" of the overall operation, calling remote functions and gathering data from various sources in one place to determine what to do next.

2.2.1 Initiation

The orchestrator listens for jobs on the queue. When it sees a job, that means there is a turn it needs to process.

2.1.2 Remote Procedure Calls

The orchestrator calls the camera interface to get the ID of the video, and then passes that ID to the image processing server in order to get back data from the board/pieces

2.1.3 Game State Tracking and Reasoning

Once the orchestrator has the piece data, it applies a wide array of logical reasoning in order to accurately determine the move that was made and engulfs the new data with the current game state. Once it has this, it uses Stockfish to determine a next move and then tells the robot interface to make that move.

2.3 Robot Interface

Purpose:

Once we interpret the board and determine the next move, we need a way to tell the robot to physically make that move. This is the job of the robot interface

2.3.1 Initiation

The robot interface is a simple HTTP server with one endpoint. The orchestrator will call it with a “from square” and “to square” (an empty to square to move the piece off the board).

2.3.2 Joint Calculation

The robot interface will use forward and inverse kinematics in order to calculate the joint angles necessary to go above the square, go straight down, go straight up, and move to the desired destination.

2.4 MuJoCo Simulator

Purpose:

The simulator allowed us to test inverse kinematics and pick place with the robot when we didn’t have access to the lab and during days when it was closed.

Implementation:

The XML for the robot/gripper combination was provided by our client, and after some adjusting we got it running in the simulator.

We modified the XML to add in cube objects for the robot to pick up. We designed inverse kinematics which used the robots information and the cubes information to simulate moving over the piece, and picking it up. Eventually we were able to simulate actions such as captures.

The way the code works is that when you run it, it opens the simulation executable. Then the robot moves based on the actions put in the main function until it is finished.

2.5 Kinematics

Purpose:

Kinematics are very important to get the robot in the right place. Forward Kinematics find the location and orientation of the wrist center. Inverse kinematics are what calculates the joint angles the robot needs to be at. This allows the robot to move in the proper place to pick up pieces. It also prevents the robot arm from moving in a way that would damage the arm (trying to reach a location by going under the table for example).

Implementation:

There were multiple solutions we considered throughout the project, but we landed on a 3-joint solution that worked for our needs.

Forward Kinematics: In our solution we get a matrix that describes the rotation and position of each of our first 3 joints: shoulder_pan, shoulder_lift and elbow. We multiply these matrices together to find the current orientation and location of the wrist center relative to the robot base. This is vital as the end goal is to put the center of the wrist directly over the piece.

Inverse Kinematics: This is what gives us the joint angels for our first 3 joints that they need to be in. We input the x,y,z target location and calculate the shoulder_pan rotation so it points towards the target. Next we calculate the distance to the target from the robot base and the shoulder joint. We then use trigonometry to calculate the target joint angles for shoulder_lift and elbow.

From this we calculated the joint positions at every square. When moving to pick up a piece the process is as follows. We first return the first 3 joints from the saved position of the target square and use them in forward kinematics to convert the square into a XYZ target for the center of the gripper. Next we use inverse kinematics to calculate the new joint angles for the shoulder_pan, shoulder_lift and elbow, and use the saved wrist joints to make sure the gripper is facing down and in the correct position. We hover down and pick up the piece, move to a safe location and then repeat on the square we plan on moving to.

Chapter 3: Work Done

3.1 Individual Contributions

Aiden Streicher:

- Primary contact for the client throughout the project.
- Labeled images for recognition model
- Wrote scripts to generate image recognition and classification models + created multiple models.
- Improved image recognition by getting x,y coordinates and checking/removing duplicates
- Set up MuJoCo simulation environment, including robot and interactable objects.
- Created working Inverse Kinematics and simulated pick and place inside MuJoCo for all types of moves (move, capture, castle)
- Wrote helper functions to help streamline piece movement and allow functional castling.
- Cut up and uploaded the final demo video.

Benjamin Johll:

- Image gathering & labelling for models
- Third-party dataset collection & format-conversion script
- Image recognition & classification model training
- Integration code for image recognition
- Robotics simulation setup
- Chess-to-robotics edge case helper functions

Andrew Drost

- Researched methods for board detection
- Designed a robust pipeline for line detection, perspective warping, and piece mapping
- Implemented a machine learning algorithm for detecting chess board corners
- Created an image difference utility
- Made a camera calibration script

- Designed the gRPC prototypes and implemented the gRPC server in python
- Researched prototyped forward and inverse kinematics used in final product
- Set up docker containers for the computer vision aspects of the project

Janus Golczewski

- Created the GUI, BFF, orchestrator, camera interface, and machine learning pipeline
- Created robot interface used in final product
- Singlehandedly wrote the code for chess piece detection used in final product
- Setup the initial components, set up docker for many components including the auxiliary services like the DB, RabbitMQ, MinIO
- Singlehandedly contributed over half of the labeled images/training data
- Trained all the models used in the final product
- Assisted teammates with researching and deciding on solutions
- Refactored significant portions of the codebase after early failures to give teammates a more solid foundation to work off of
- Researched prototyped forward and inverse kinematics used in final product

Chapter 4: Results

4.1 Overview

This semester we successfully created a working image recognition system, and successfully played chess with the robot arm making moves.

Our client is running this project again next semester in an attempt to polish the robot side. As such we handed over our image recognition software to help next year's groups get a jump start on the project and give them more time to work on the robot. We also have the simulator and ROS code on our gitlab.

4.2 Specific Milestones

Throughout the semester we achieved a number of milestones:

- Created a chess game GUI and integrated stockfish as an opponent.
- Created an image recognition and image classification model that successfully identifies pieces with 90-95% accuracy depending on the piece.
- Created software that successfully recognizes the individual squares on the chess board, allowing for FEN's to be generated from a board state.
- Completed initial pipeline of turning a board state into a FEN for a chess game and getting stockfish moves in return.
- Set up MuJoCo simulation environment and successfully simulated inverse kinematics and pick and place within the environment.
- Calculated forward and inverse kinematics on the robot.
- Implemented pick and place on the real world board, picking up a piece and placing it in a new target location.
- Integrated everything into a final pipeline, successfully playing chess against stockfish controlled by a robot arm.

Appendix- Demo Slides

Team 8

Project Joseph Hoane



Objectives & Purpose

- Use an image recognition model to recognize a chessboard
- Command robot arm to play best moves from an engine
- For use in studying & researching robotics
- Improve upon previous teams' attempts

Design Constraints

- Computer, robot, pieces, board provided by client
- Limited camera angles
- Sequential development

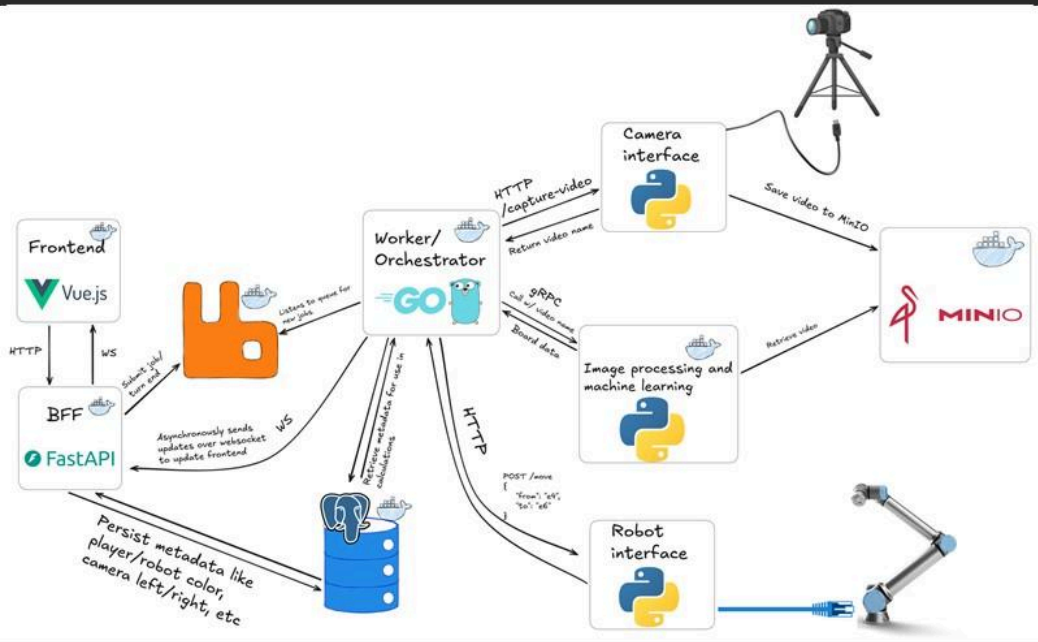
Scope Limitations

- Manual grip control
- Simple capturing
- Promotions

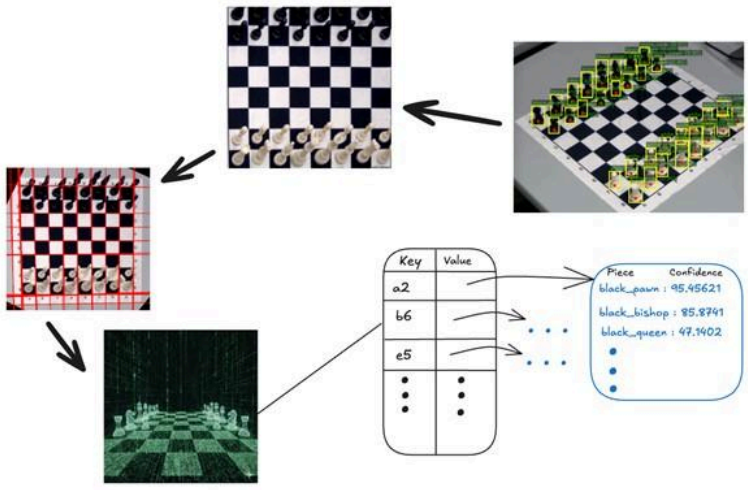
Working With Chess

- Using Stockfish
- Chess Edge Cases





Board Detection Pipeline

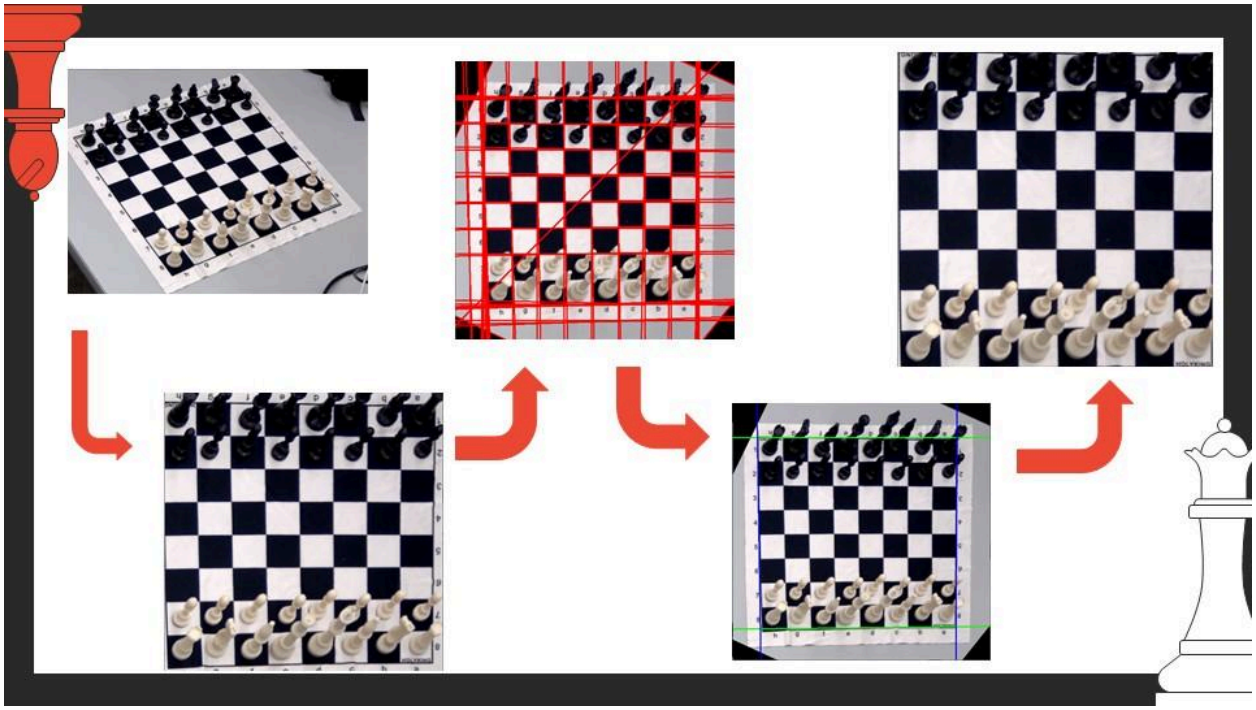


Beginning of each move

- User will move their piece and click "start game as B/W" or "end turn"
- Frontend will send an HTTP request to the BFF
- BFF will add a job the to the RabbitMQ queue
- The worker listens for new messages on the queue and orchestrates the process below:

Record video ⇨ Run models for piece and corner detection

⇨ Warp the board image and detected piece locations



Camera Calibration

- Not easy to tell how the warp will function without trial and error
- Calibration script makes it super easy

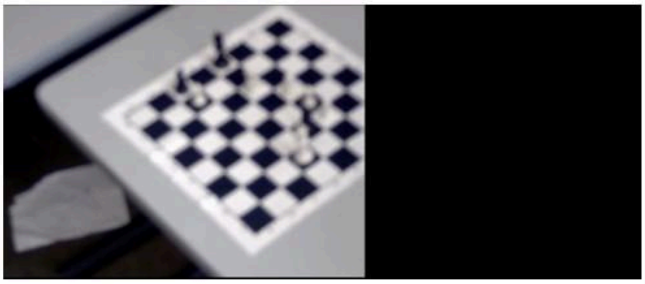
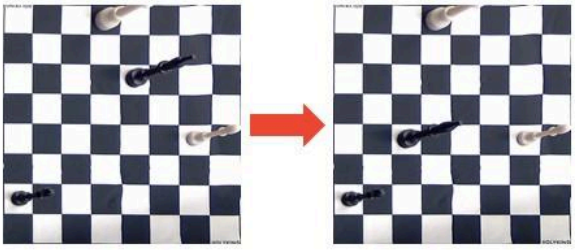
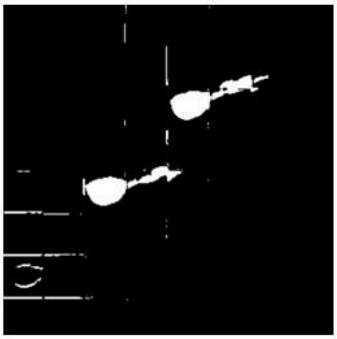


Image Difference

- Sometimes piece detection misses something
- Pieces are occasionally mislabeled
- We can compare current board to previous to find what piece moved





Testing

- Each step in the image processing pipeline has many different parameters
- Adjusting parameters for one case may mess up others
- Test scripts ensure pipeline is functioning for all expected scenarios


```
{
    "image": "case_05",
    "camera_right": true,
    "robot_white": false,
    "pieces": [
        {"name": "black_queen", "x": 515, "y": 201},
        {"name": "black_knight", "x": 293, "y": 127},
        {"name": "white_king", "x": 410, "y": 186},
        {"name": "white_pawn", "x": 277, "y": 199},
        {"name": "white_knight", "x": 454, "y": 284},
        {"name": "white_queen", "x": 211, "y": 207},
        {"name": "black_bishop", "x": 342, "y": 269},
        {"name": "black_rook", "x": 362, "y": 324},
        {"name": "white_bishop", "x": 248, "y": 309},
        {"name": "black_pawn", "x": 291, "y": 332},
        {"name": "white_rook", "x": 426, "y": 400},
        {"name": "black_king", "x": 132, "y": 295}
    ],
    "detected_corners": [
        [260,67.5],
        [605,206.25],
        [415,468.75],
        [55,277.5]
    ],
    "fen": "6q1/1n2K3/8/2P3N1/1Q2b3/5r2/3Bp2R/1k6"
},
```


```
testing\test_diff.py ............
testing\test_warp.py ..F.......

================================ FAILURES ================================
_____________________ test_warp[case2] _____________________

case = {'camera_right': True, 'detected_corners': [[315, 180], [425, 255], [270, 318.75], [160, 213.75]], 'fen': '8/7B/RqQ5/3
```

Upon receiving chessboard data

- Machine Learning outputs are fundamentally non-deterministic in contrast to the very "black and white" game of chess
- What measures can be taken to mitigate unreliable outputs as much as possible?

Techniques to improve robustness pre and post model output

2 principles:

- Ample data aggregation
- Logical reasoning

Ample data aggregation

- Problems:
  - Flickering ⇨
  - Differing outputs



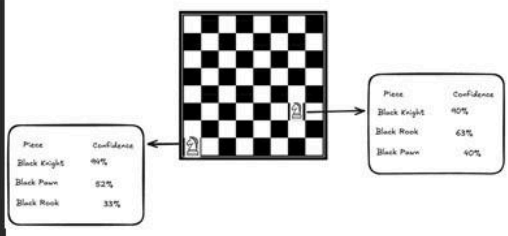
- Solution?
  - Always aggregating and averaging data that originated from ML output

Instead of running the model once on a single thread, run the model on every frame of a short video and average results

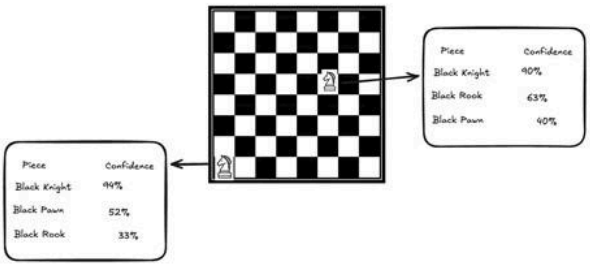
Ample data aggregation cont.

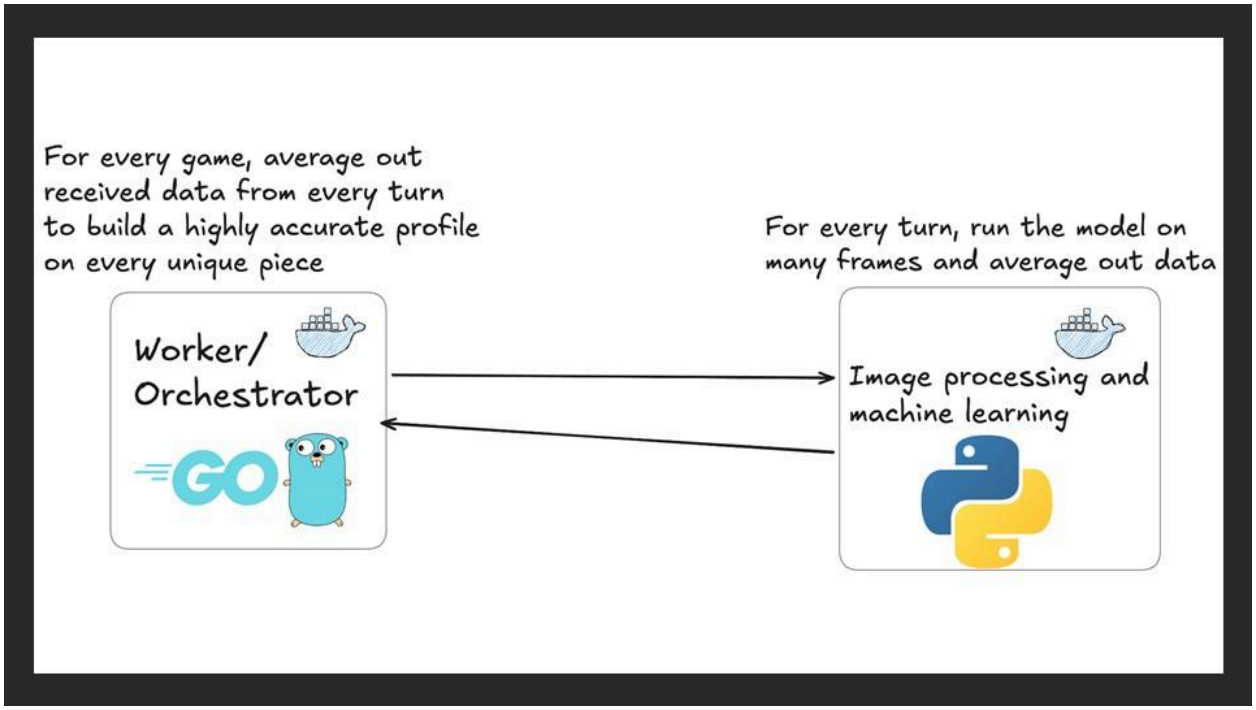
- We can aggregate many model scans on a per-turn basis, but we can also aggregate across the entire game

Move x



Move x+1





Logical reasoning

- How can we make the program be able to tell when something went wrong and make the most logical move to at least “survive till next scan”?

- Primitives:
  - For every move (besides castling), the square the piece moved from becomes empty, the square the piece moved to either becomes goes from empty to full or from B -> W or W -> B (depending on the color of the piece that moved)

Detection cases and how the software can recover

- Case: Detecting 0 squares moved to OR 0 squares moved from OR both
  - One of the worst cases, we can have no good idea of how the player changed the game state, robot makes a move based on the state of the board we *do* know.

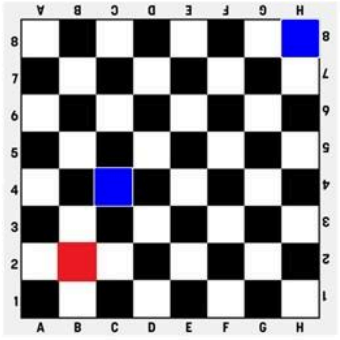
- Case: 1 square moved from and 1 square moved to
  - Ideal - this is what should happen

Detection cases and how the software can recover cont.

- Case: More than 1 piece moved from OR more than 1 piece moved to detected
  - Initiate the recovery pipeline

= Detected candidate "moved from" square

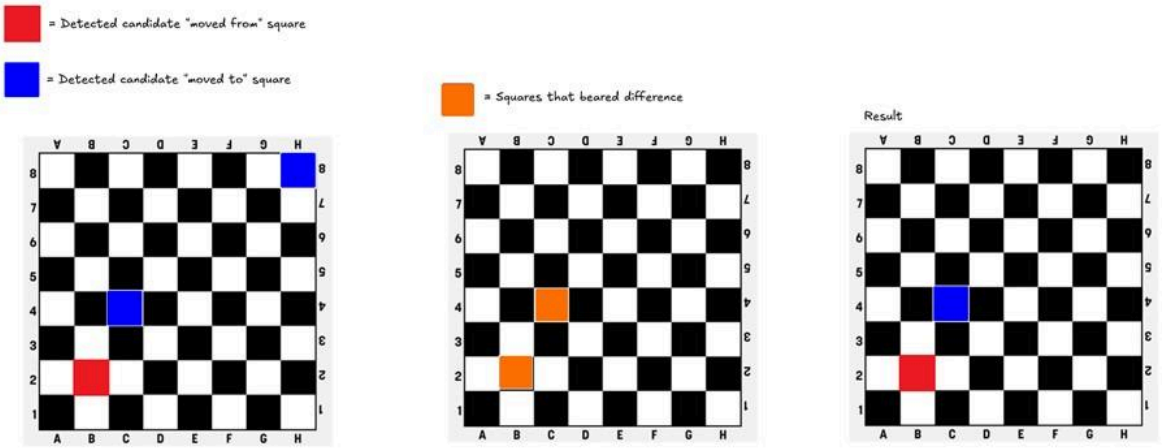
= Detected candidate "moved to" square



Recovery pipeline step 1 - use image diff if available

- We can work without the image diff function but if we have it, use it
- Image diff is a gRPC call to the image processing call and returns squares that show the squares that had the highest amount of difference between move X and move X+1
- Simply filter out detected squares we have that don't align with image diff

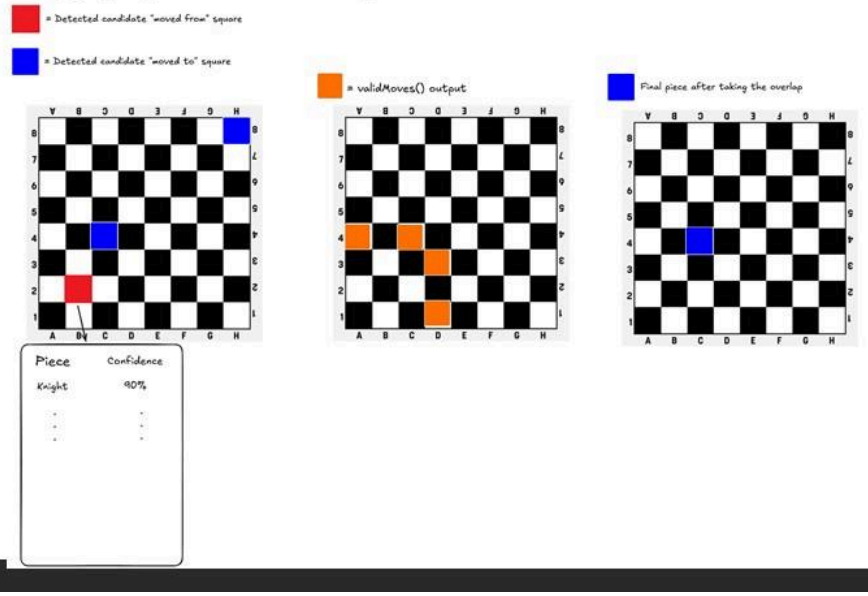
Image diff example



Recovery pipeline step 2 - apply chess logic

- For each “moved from” square, the program asks the hypothetical, “what if this one was it?”
  - Look at what the software thinks the piece is at the “from square” i.e knight 90%
  - Utilize a chess library’s function determinveValidMoves() to get all the moves the player could have possibly made
  - Look for pairs between the detected “to squares” and the valid moves, if match, that move is a valid possibility

Recovery pipeline step 2 visualized



Additional application of logical reasoning

- Once the from and to squares have been determined, cross reference with our piece confidences to see if it makes sense
- For example, if the piece moved from a4 to h4, penalize the confidences of all the pieces that could not have possibly moved from a4 to h4

Robot interface

- Fundamentally, we had a way to see data from the robot and to command the robot with joint angles via ROS
- Joint angles are not easy to reason with, difficult to make robot correctly go to the proper height or from different squares without using very many hardcoded values
- In order to be able to go between joint angles (hard to reason with) and XYZ coordinates (easy to reason with) we used forward and inverse kinematics

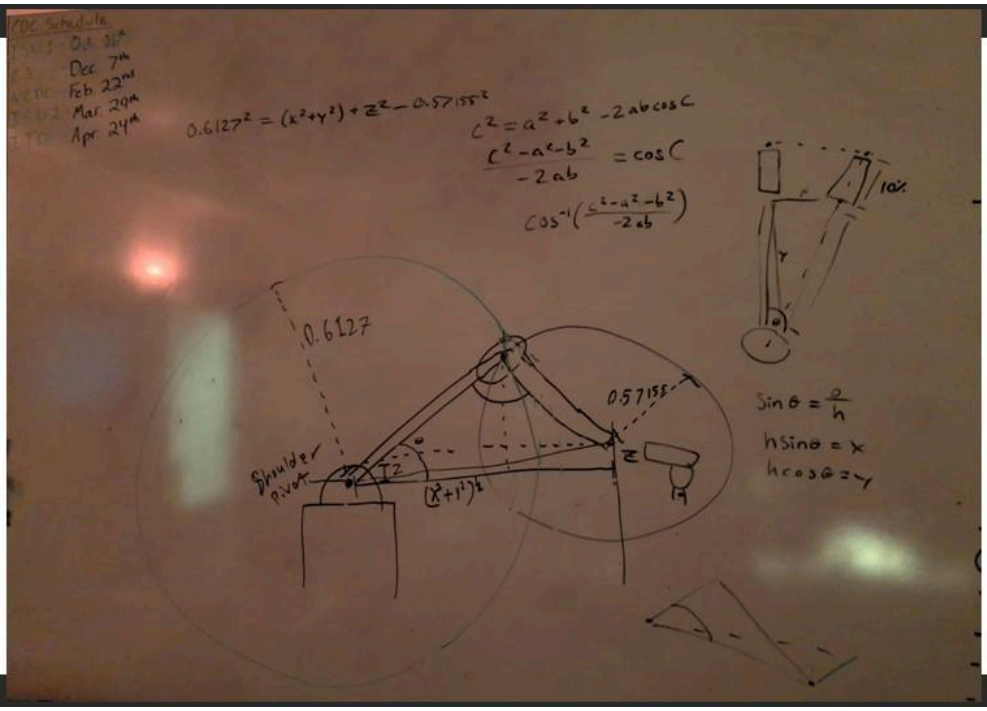
Kinematics

- Forward kinematics is the conversion of joint angles -> XYZ
- Inverse kinematics is the conversion of XYZ -> joint angles

- First we programmed functions to easily switch between these two representations in our robot interface code

- Surprisingly unclear guides/documentation out there on calculating inverse kinematics, had to follow some advice from an old thread and do the math ourselves for 3 joint IK

https://forum.arduino.cc/t/inverse-kinematics/676888/18



Programmatically interacting with ROS/the robot

- Our environment required for us to use ROS 1 which is considerably outdated (required for us to use a linux distro version 5 years out of date to run)
- The build system, catkin, introduced a lot of complexity for our relatively simple needs
- After experimenting with catkin, we found that all the SDK that comes with catkin really did is provide an interface to the same commands we can send over terminal

Creating a minimalist ROS interface

- In order to bypass having to use the outdated software stack of ROS 1 and the complexity it brought, we used a simple work around in our robot interface
- Simple HTTP server that listens for a request with a body like { from: "b2", to: "g5"} and then uses the kinematics functions we made to calculate joint angles, then sends the move command with those angles directly to the terminal

Example command the robot interface sends

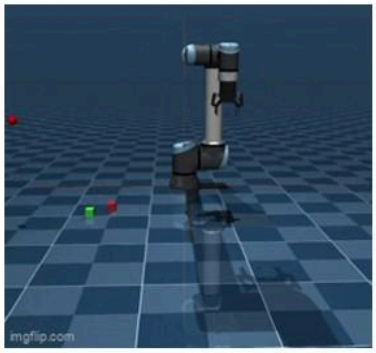
rostopic pub -1 /scaled_pos_joint_traj_controller/follow_joint_trajectory/goal control_msgs/FollowJointTrajectoryActionGoal "{goal: {trajectory: {joint_names: ['**elbow_joint**', '**shoulder_lift_joint**', '**shoulder_pan_joint**', '**wrist_1_joint**', '**wrist_2_joint**', '**wrist_3_joint**'], points: [{positions: [*-1.3117430210113525*, *-2.147092958489889*, *0.020304590463638306*, *-1.2039109331420441*, *-4.647023979817526*, *3.436668872833252*], time_from_start: {secs: 4}]}}}"

MuJoCo

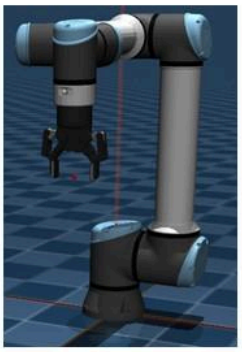
MuJoCo

- Robot Simulation Software
- Uses the same robot/gripper as real life
- Added cubes for testing

- Successes:
  - Implemented inverse kinematics
  - Implemented pick and place
  - A good relic to hand over



What went wrong

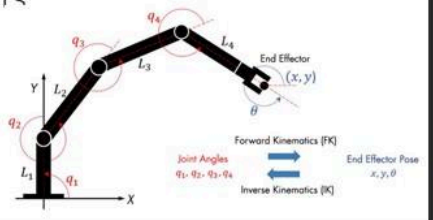


- Failed to majorly affect the actual robot production
- Failed to convert inverse kinematic solution to something useable

- In Hindsight:
  - Still well functioning
  - Should have focused more on how to make the real robot better
  - Spent too much time working on simulator functionality

Major Problem- Inverse Kinematics

- 6 joint angles to get the robot in the position we want
- Input is x, y, z location
- Need to calculate
- Avoids dangerous or impossible movements



Solution 1- IK Fast

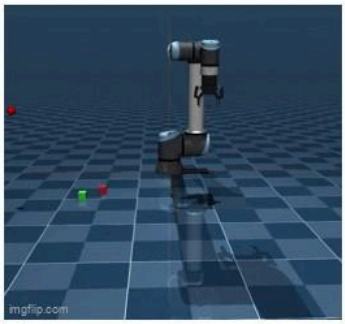


- Analytical Solver used in ROS environments
- Pros:
  - Easy to implement
  - Very accurate
  - Works with our robot
- Cons:
  - Requires import. which is not possible in our environment

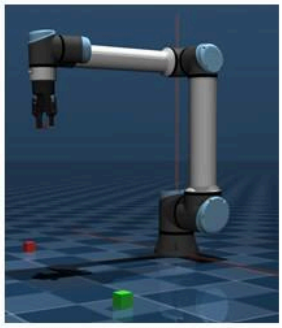
Solution 2- MuJoCo Solution

The Solution:

- Use Jacobian-based damped least-squares
- Uses many iterations to safely reach target position
- Slowly moves over each iteration
- Developed and functional in MuJoCo environment



Solution 2- Jacobian-based damped least-squares



Pros:

- Good accuracy
- 6 joint solution
- Very safe

Cons:

- Comparatively slow
- Implementation relied on some MuJoCo values (harder to transfer over)
- Ran out of time to fully translate over

Solution 3- Geometric Closed-Form

- Uses trigonometry and distances to calculate angles
- Does it in 1-shot compared to over iterations
- Calculates 3 joint angles (shoulder_pan, shoulder_lift, elbow_angle)

Pros:

- It works in our ROS environment
- Is accurate and consistent
- Very fast

Cons:

- Only calculates 3 joint angles
- Worse at handling singularities
  - When the path to a location creates unsafe joint configurations

Using this solution

We calculated the 6 joint angles for every square and used them:

- First we use forward kinematics from the first 3 angles to get an XYZ target
  - Because joint angles are over target square gripper center is over the square

- Next calculate inverse kinematics on this XYZ target to make sure joint angles are good
- Use saved wrist values so gripper is in the correct orientation
- Drop down to successfully pick up the piece

Accomplishments

Major:

- Created a successful image recognition and classification model.
- Turned a video of the board into a FEN string.
- Calculated kinematics for the robot and implemented pick and place.
- Played chess against the robot!

Minor:

- Created a chess GUI and implemented stockfish as an opponent.
- Built upon our past image recognition models to continuously improve it.
- Set up MuJoCo simulation environment and implemented both inverse kinematics and pick and place.

Individual Contributions

Aiden Streicher:

- Wrote script to create image recognition and classification models.
- Implemented MuJoCo simulation
- Created a variety of helper functions including for piece movement.

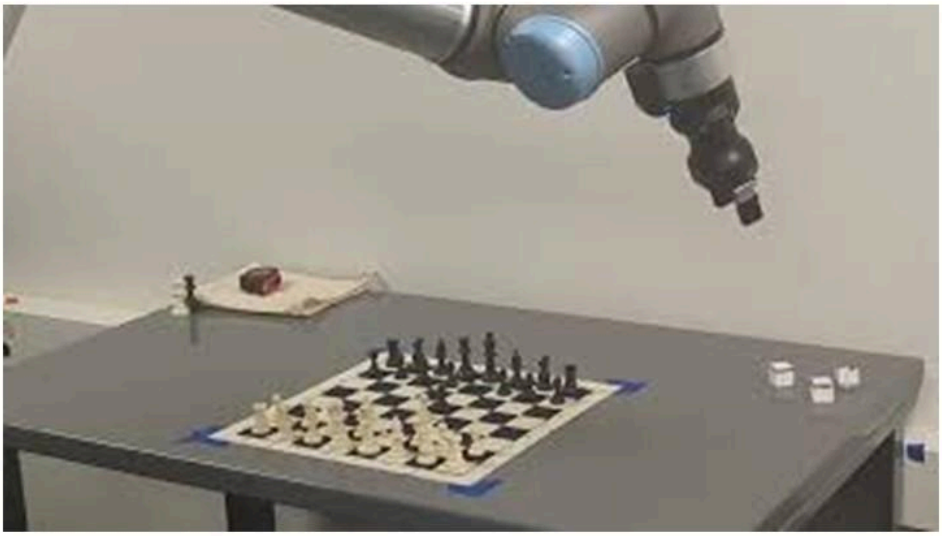
Andrew Drost:

- Chess board perspective warp
- Image processing and line detection
- Corner detection
- gRPC prototypes and python server
- Forward kinematics
- 3 joint IK
- Testing scripts

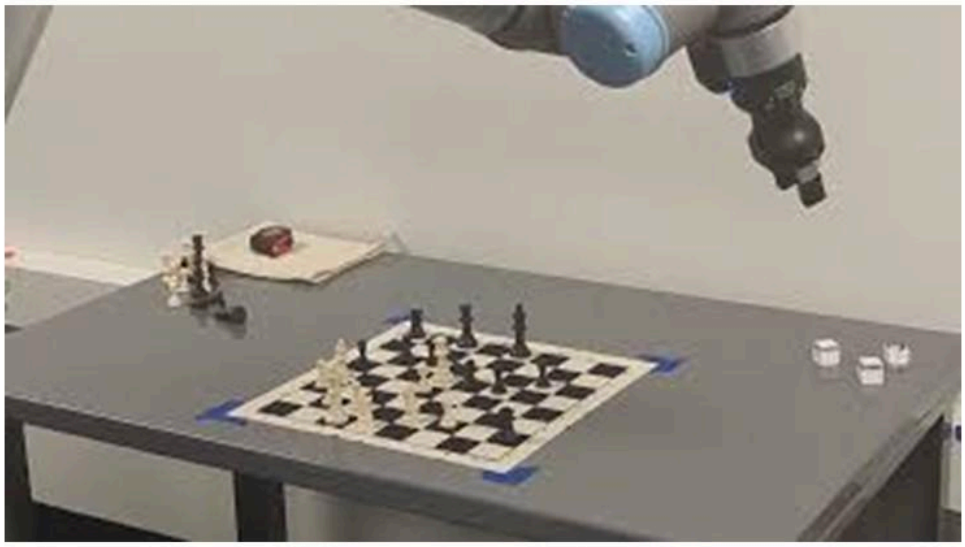
Benjamin Johll

- Image gathering & labelling
- External dataset gathering & integration script
- Image recognition & classification model training
- Image model integration
- Chess-to-robot edge-case handling

Demo Video 1



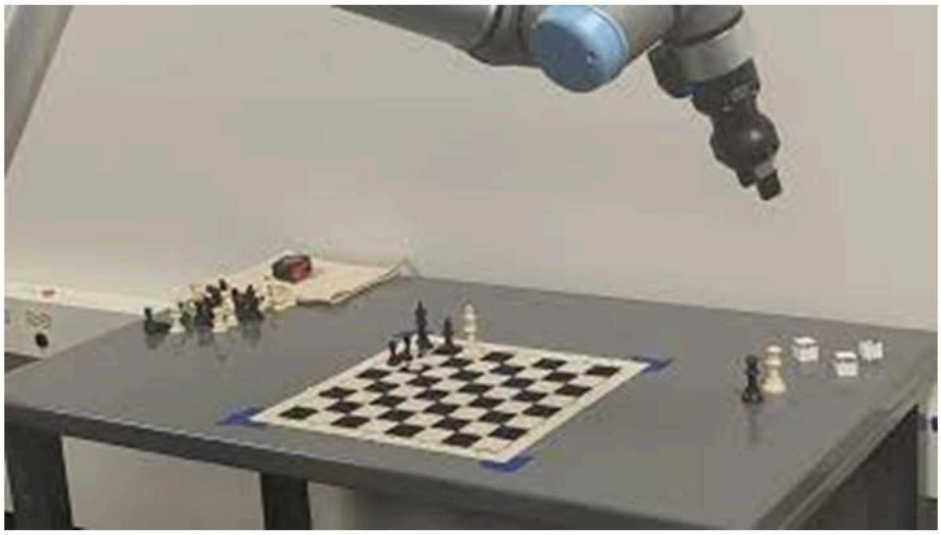
Demo Video 2



Demo Video 3



Demo Video 4



Thanks!

Questions?

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution